Multiagent Systems: Coursework 2018

Edward Evans

November 2018

Contents

1 Vehicle Implementation

1.1 Vehicle

```
1  public Vehicle(Location l){
2     super(l);
3     this.carryingSample = false;
4  }
```

The `Vehicle` method's purpose is to construct a new instance of the Vehicle class. The Vehicle class extends `Entity`, as a result Vehicle's constructor must call the super method. This method takes one parameter of type `Location`, this is used to set the initial location of the vehicle on the map. This constructor also initializes the value of `carryingSample` to `false`, this means that the vehicle when created is not in possession of any rocks.

1.2 act

```
1  public void act(Field f, Mothership m, ArrayList<Rock> rocksCollected) {
2     actCollaborative(f,m,rocksCollected);
3     // actSimple(f,m,rocksCollected);
4  }
```

The `act` method, acts as an interface for other classes to call. This method takes in three parameters, these include information about the field which the simulation is taking place upon, information about the Mothership that can be located in the field and a list of rocks to be collected. Line 2 or 3 can be called, depending on the mode selected.

1.3 actCollaborative

```
1  public void actCollaborative(Field f, Mothership m, ArrayList<Rock>
       rocksCollected){
2     // if carrying a sample and at the base then drop sample (1)
3     if (this.carryingSample && f.isNeighbourTo(this.getLocation(),
          Mothership.class)) {
4        // Drop sample
5        this.carryingSample = false;
6        return;
7     }
8
9     // if carrying a sample and not at the base then drop two crumbs and
          travel up gradient (5)
10    if (this.carryingSample && !f.isNeighbourTo(this.getLocation(),
          Mothership.class)) {
11       // Drop two crumbs
```

```
12          f.dropCrumbs(this.getLocation(), 2);
13
14          // Travel up gradient
15          this.moveUpGradient(f);
16          return;
17      }
18
19      // if detect a sample then pick sample (3)
20      if (f.isNeighbourTo(this.getLocation(), Rock.class)) {
21          // Pick up sample
22          this.pickupRock(f, rocksCollected);
23          return;
24      }
25
26      // if sense crumbs then pick up 1 and travel down gradient (6)
27      if (f.getCrumbQuantityAt(this.getLocation()) > 0) {
28          // Pick up crumb
29          f.pickUpACrumb(this.getLocation());
30          // Travel down gradient
31          this.moveDownGradient(f);
32          return;
33      }
34
35      // if true then move randomly (4)
36      if (true) {
37          // move randomly
38          this.moveRandomly(f);
39          return;
40      }
41  }
```

The method `actCollaborative` is called by `act`, when the simulation requires the vehicles to work collaboratively. This method takes in the same parameters as `act`.

This method is constructed using a set of `if` statements, these statements are based on a set of actions that should be performed, if the preconditions are met. The order in which these rules are executed is based on a hierarchy. In this method the rules are as follows:

1. if carrying a sample and at the base then drop sample
2. if carrying a sample and not at the base then drop two crumbs and travel up gradient
3. if detect a sample then pick sample
4. if sense crumbs then pick up 1 and travel down gradient
5. if true then move randomly

The first rule is implemented in lines 3-7, the precondition of this rule requires us to know if the vehicle is carrying a sample, this information is stored by the class in a variable `carryingSample`, therefore we can check if the vehicle is holding a rock by seeing if this variable equates to `true`. The other precondition for this rule is that the vehicle is next to the mothership, this can be checked by using the method `isNeighbourTo` attached to the field class. `isNeighbourTo` is called with two parameters, in this occurrence the location of the vehicle and the class `Mothership` are passed in. The result of this method would then be `true` if the vehicle was next to the mothership. If both the preconditions are met the vehicle needs to drop the sample, this is achieved by setting the variable `carryingSample` to `false`.

The next rule is implemented in lines 10-17 and has one matching precondition to rule 1, that the vehicle is carrying a sample. The second precondition is that the vehicle is not adjacent to the mothership. The second precondition can be achieved in a similar way to the first rule, by adding the prefix `!`(not) to the same method as before. If this rule's preconditions are met the vehicle should drop 2 crumbs and should travel up gradient. The field class contains a method `dropCrumbs` which can be passed the location of where to drop a certain quantity of crumbs. As seen on line 12, the method is passed the current location of the vehicle and a quantity of 2. The next action for the vehicle is to travel up gradient, this is implemented in the method `moveUpGradient` which is explained later in the report.

The third rule uses the same method as the first two, `isNeighbourTo` to check if there is an entity of type `Rock` adjacent to the vehicle. If this condition is true, the method `pickUpRock` is called, the explanation of this method can be found within the report.

The penultimate rule of `actCollaborative` is implemented in lines 27-31, the actions for this rule will occur if the vehicle detects there are crumbs in it's current position. To check if there are crumbs at the vehicle's position the method `getCrumbQuantityAt` from within the field class can be used. By checking to see if there are more than zero crumbs at the current position. If that turns out to be the case, the method `pickUpACrumb` is called, followed by the method `moveDownGradient` which is explained within this report.

The last rule is implemented in lines 36-40, this rule will always trigger if any previous rules do not. The action for the vehicle is to move randomly this is implemented in the method `moveRandomly`.

1.4 actSimple

```
1  public void actSimple(Field f, Mothership m, ArrayList<Rock>
       rocksCollected){
2    // if carrying a sample and at the base then drop sample (1)
3    if (this.carryingSample && f.isNeighbourTo(this.getLocation(),
         Mothership.class)) {
4      // Drop sample
```

```
5         this.carryingSample = false;
6         return;
7      }
8
9      // if carrying a sample and not at the base then travel up gradient
           (2)
10     if (this.carryingSample && !f.isNeighbourTo(this.getLocation(),
           Mothership.class)) {
11        // travel gradient
12        this.moveUpGradient(f);
13        return;
14     }
15
16     // if detect a sample then pick sample (3)
17     if (f.isNeighbourTo(this.getLocation(), Rock.class)) {
18        // Pick up sample
19        this.pickupRock(f, rocksCollected);
20        return;
21     }
22
23     // if true then move randomly (4)
24     if (true) {
25        // move randomly
26        this.moveRandomly(f);
27        return;
28     }
29  }
```

`actSimple` is called by `act` and is an alternative to `actCollaborative`. This method is very similar to its alternative, with the difference being that vehicles do not utilise crumbs. The rules used by this method are as follows:

1. if carrying a sample and at the base then drop sample
2. if carrying a sample and not at the base then travel up gradient
3. if detect a sample then pick up sample
4. if true then move randomly

All these rules are almost exactly the same as the ones found in `actCollaborative`. The differences being that "if sense crumbs then pick up 1 and travel down gradient" has been removed and that the second rule does not drop any crumbs whilst traveling up gradient.

1.5 getLocationUpGradient

```
1  public Location getLocationUpGradient(Field f) {
2     ArrayList<Location> freeSpaces =
          f.getAllfreeAdjacentLocations(this.getLocation());
```

```
3
4     int highestSignal = f.getSignalStrength(this.getLocation());
5     Location highestSignalLocation = this.getLocation();
6
7     for (Location currentSpace : freeSpaces) {
8        if (f.getSignalStrength(currentSpace) > highestSignal) {
9           highestSignal = f.getSignalStrength(currentSpace);
10          highestSignalLocation = currentSpace;
11       }
12    }
13    return highestSignalLocation;
14 }
```

This method is responsible for finding a free adjacent location to the current location of the vehicle with a higher signal strength. To get the location, firstly all the free adjacent spaces are found using a method from the field class. Then variables which track the highest signal strength are initialised to the current location of the vehicle. The method then proceeds to iterate over the free spaces calculating the signal strength of each space and checking to see if it is higher than the previously seen highest value. Once all of the spaces have been checked, the value of the variable `highestSignalLocation` should contain an adjacent space with a higher signal value than the current location of the vehicle. This variable is then returned to where the method was called.

1.6 moveUpGradient

```
1 public void moveUpGradient(Field f) {
2    Location nextLocation = this.getLocationUpGradient(f);
3    if (nextLocation != null) {
4       this.updateLocation(f, nextLocation);;
5    }
6 }
```

This method moves the vehicle to a free space adjacent to the current location of the vehicle with a higher signal strength. To achieve this the method `getLocationUpGradient` is called and the result is passed into `updateLocation`.

1.7 getLocationDownGradient

```
1 public Location getLocationDownGradient(Field f) {
2    ArrayList<Location> freeSpaces =
         f.getAllfreeAdjacentLocations(this.getLocation());
3
4    int lowestSignal = f.getSignalStrength(this.getLocation());
5    Location lowestSignalLocation = this.getLocation();
6
```

```
   for (Location currentSpace : freeSpaces) {
      if (f.getSignalStrength(currentSpace) < lowestSignal) {
         lowestSignal = f.getSignalStrength(currentSpace);
         lowestSignalLocation = currentSpace;
      }
   }
   return lowestSignalLocation;
}
```

The object of this method is to find a free space adjacent to the current location of the vehicle with a lower signal strength. To return the location, the method `getAllfreeAdjacentLocations` is called from the field class. Then variables are initialised to the location and signal strength of the vehicle's current location. After this the free spaces are iterated over and each of their signal strengths are calculated and checked against the lowest currently stored. After checking all the spaces the variable `lowestSignalLocation` will contain the location of an adjacent space with the lowest signal strength, which can then be returned.

1.8 getLocationDownGradientCrumbs

```
public Location getLocationDownGradientCrumbs(Field f) {
   int signalWeight = 1;
   int crumbsWeight = 10;
   ArrayList<Location> freeSpaces =
       f.getAllfreeAdjacentLocations(this.getLocation());

   int bestRating = (signalWeight *
       f.getSignalStrength(this.getLocation())) -
    (crumbsWeight * f.getCrumbQuantityAt(this.getLocation()));
   Location bestRatingLocation = this.getLocation();

   for (Location currentSpace : freeSpaces) {
      int currentRating = (signalWeight *
          f.getSignalStrength(currentSpace))
       - (crumbsWeight * f.getCrumbQuantityAt(currentSpace));
      if (currentRating < bestRating) {
         bestRating = currentRating;
         bestRatingLocation = currentSpace;
      }
   }
   return bestRatingLocation;
}
```

This method is very similar to `getLocationDownGradient`, it has the same function with one variation. Instead of just using the signal strength to decide which adjacent space to return, this method takes into account how many crumbs are stored in that location. When iterating over all the free adjacent spaces to the

current location of the vehicle, a rating is calculated.

```
rating = (signalWeight * signalStrength)- (crumbsWeight * noOfCrumbs)
```

This rating is then checked against the previous lowest rating. The two weights used to calculate the rating, can be changed depending on the importance of the corresponding variable. The space with the lowest rating after checking all the spaces is then returned.

1.9 moveDownGradient

```
1 public void moveDownGradient(Field f) {
2   Location nextLocation = this.getLocationDownGradient(f);
3   // Location nextLocation = this.getLocationDownGradientCrumbs(f);
4   if (nextLocation != null) {
5     this.updateLocation(f, nextLocation);;
6   }
7 }
```

This method moves the vehicle to a free space adjacent to the current location of the vehicle, with a lower signal strength. To achieve this the method `getLocationDownGradient` is called and the result is passed into `updateLocation`.

This method can be altered by using line 3 instead of 2. With this alteration the vehicle's next position will be determined by the number of crumbs in adjacent spaces as well as the signal strength.

1.10 pickupRock

```
1 public void pickupRock(Field f, ArrayList<Rock> rocksCollected) {
2   Location rockLoc = f.getNeighbour(this.getLocation(), Rock.class);
3   if (rockLoc != null) {
4     Rock newRock = new Rock(rockLoc);
5     rocksCollected.add(newRock);
6   }
7   this.carryingSample = true;
8 }
```

This method checks to see if a rock is adjacent to the vehicle and picks the rock up. The method `getNeighbour` is called to get the location of a rock adjacent to the vehicle. The location is then used to create a rock to add to the array `rocksCollected`. Once the rock is added to that array, the simulation will later remove that rock from the field. The final thing this method does is update the variable `carryingSample` to `true`.

1.11 moveRandomly

```
1 public void moveRandomly(Field f) {
```

```
2     ArrayList<Location> freeSpaces =
          f.getAllfreeAdjacentLocations(this.getLocation());
3     Random rand = new Random();
4     int n = rand.nextInt(freeSpaces.size());
5     this.updateLocation(f, freeSpaces.get(n));
6  }
```

When this method is called the vehicle will be moved to a random free space adjacent to its current location. This is done by generating a random number between zero and the number of free spaces adjacent to the vehicle. This is then used as an index to get the location of the space to move to, using the `updateLocation` method.

1.12 updateLocation

```
1  public void updateLocation(Field f, Location newLocation) {
2     f.clearLocation(this.getLocation());
3     this.setLocation(newLocation);
4     f.place(this, newLocation);
5  }
```

This method moves the vehicle to a specified location. To move the vehicle, a new location has to be given, then the space the vehicle is currently occupying has to be cleared. The vehicle can then be placed on the field in the new location and the vehicle's variable storing its location can be updated to the new location.

2 Improvements

2.1 Sense Crumbs

An improvement can be made when a vehicle is searching for a rock. When the vehicles are acting collaboratively, vehicles drop crumbs when they are returning to the mothership whilst carrying a rock, these crumbs can be used to help other vehicles navigate to new rocks.

An example of such an improvement can be seen in the method `getLocationDownGradientCrumbs`. Whilst a vehicle is attempting to find a new rock the original behaviour would be to find a new location further away from the mothership by checking to see if the signal strength to the mothership was lower, however this can be improved by checking the available locations for the amount of crumbs they contain. A rating can be assigned to the available spaces, which is calculated based on the signal strength and the amount of crumbs in that location. This rating can be calculated by subtracting the number of crumbs from the signal strength, depending on the importance of the two variables in the calculation, a weighting can be given to them. By multiplying the number

of crumbs and the signal strength by the respected weighting you can get a rating for each space which is based on the importance of each statistic. Once all the ratings have been calculated the lowest value can be found, then that would become the new location for the vehicle.

2.2 Reduce Crumbs

When the vehicles are in collaboration mode, they leave crumbs when they are carrying a sample. These crumbs are only removed when a vehicle looking for a sample picks them up. The problem with this system is trails of crumbs may lead vehicles to recently emptied cluster of rocks and the trail will only be removed once enough vehicles have followed the trail.

```
// If a crumb has been there for 1000 steps remove it
if (step % 1000 == 0 ) {
   tempField.reduceCrumbs();
}
```

This can be improved by applying a method of ageing for the crumbs, this is when a crumb will be removed after a period of time even if a vehicle has passed over it. Adding ageing for crumbs means that only trails that are being used recently will remain on the field, and trails leading to no rocks will be removed quicker. One way of achieving this is to remove a crumb from every space every x amount of steps. This has been implemented in the Simulator class within the `simulateOneStep` method by adding the code seen above.

3 Experiment

Simulator Paramaters

Simulation Parameters	
Mars Width	50
Mars Depth	50
Rock Placement	
Number of Rocks	300
Number of Clusters	7
Rock Clusters Std	2
Obstacles & Vehicles	
Obstacle	0.002

To collect the data below, the simulation was ran multiple times using a different number of vehicles each time. When running the simulation the only variables that were changed, were the number of vehicles used. The other parameters used to run the simulation were kept constant, these values can be seen in the above table.

For each variation in vehicle number, the simulation is ran with three different seeds and using the `actSimple` and `actCollaborative` methods. The results from the different seeds are then averaged.

Results using `actSimple`

Number of Vehicles	Number of Steps			
	Seed			
	133	200	10	Avg.
0.002	24926	37724	19187	27279
0.004	16414	20243	12126	16261
0.006	13258	8789	9756	10601
0.008	9775	6799	7180	7918
0.010	8520	6459	6134	7037

Results using `actCollaborative`

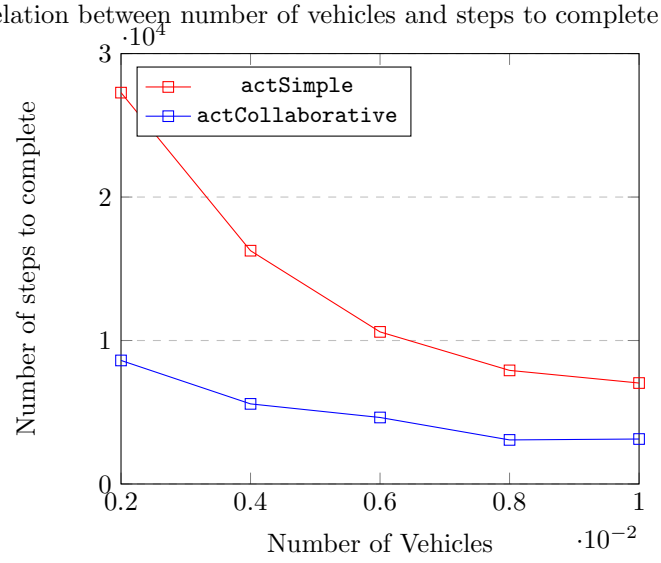
Number of Vehicles	Number of Steps			
	Seed			
	133	200	10	Avg.
0.002	6259	10052	9521	8611
0.004	5026	6250	5473	5583
0.006	5949	3917	4039	4635
0.008	2965	2600	3647	3071
0.010	3236	2411	3756	3134

From the results it can be seen that as the number of vehicles increase, the time taken for all the rocks to be collected gets smaller. Using either method this trend can be seen, this is a result of the throughput of the entire system being increased as more vehicles can be carrying samples at any one time.

It can also be seen that using the method `actCollaborative` reduces the amount of steps taken for the rocks to be collected. This result was expected, as the `actCollaborative` method implements behaviours that allow each agent to communicate and more efficiently find rocks to collect. This means that the time taken for each agent to find and pick up a new sample is reduced, therefore the time taken to collect all the samples is also reduced.

As the number of vehicles are increased the time taken reduces, however the rate at which the time is reduced seems to also reduce as more vehicles are added. This is demonstrated in the graph below, by the curve's gradient being reduced as the number of vehicles are increased. This could be a result of many things, one may be that having more vehicles in the field reduces the amount of space that each vehicle can move into, resulting in vehicles having to take longer routes to find and return rocks to the mothership.

Relation between number of vehicles and steps to complete simulation



4 Reflection

The coursework has illustrated how a simulation can be constructed and demonstrated how the order of rules which an agent has to follow are important when creating a successful simulation. It has also shown that a rule is constructed of preconditions and actions and how to implement such a rule using `if` statements.

This coursework has also shown a way in which agents can indirectly communicate with each other, in this case using crumbs. This mode of communication could be helpful when creating other multiagent systems, as it eliminates the need to create a protocol in which agents have to follow to communicate with other agents.

It has also shown how a seemingly complex behaviour can be built up from a set of simple rules. This is seen in the simulation, when each vehicle leaves a trail of crumbs which other agents follow to new rocks. Although this behaviour looks to be somewhat complex, it is actually constructed using a few very simple rules which each agent follows independently.